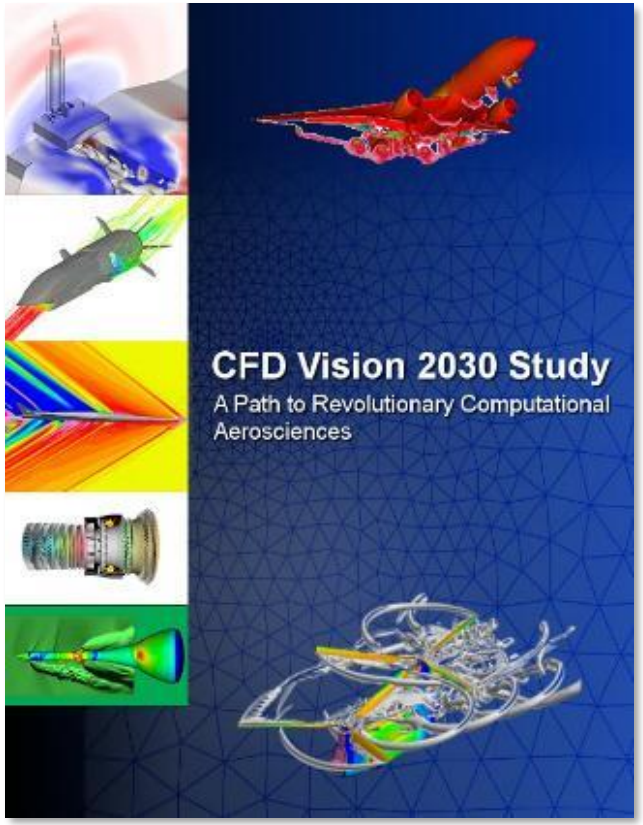
Performance Optimization Methods for a Memory-Bound, Unstructured-Grid CFD Application on Massively Parallel GPU Platforms

Christopher Stone
National Institute of Aerospace (former)

Aaron Walden Eric Nielsen Gabriel Nastac
NASA Langley Research Center

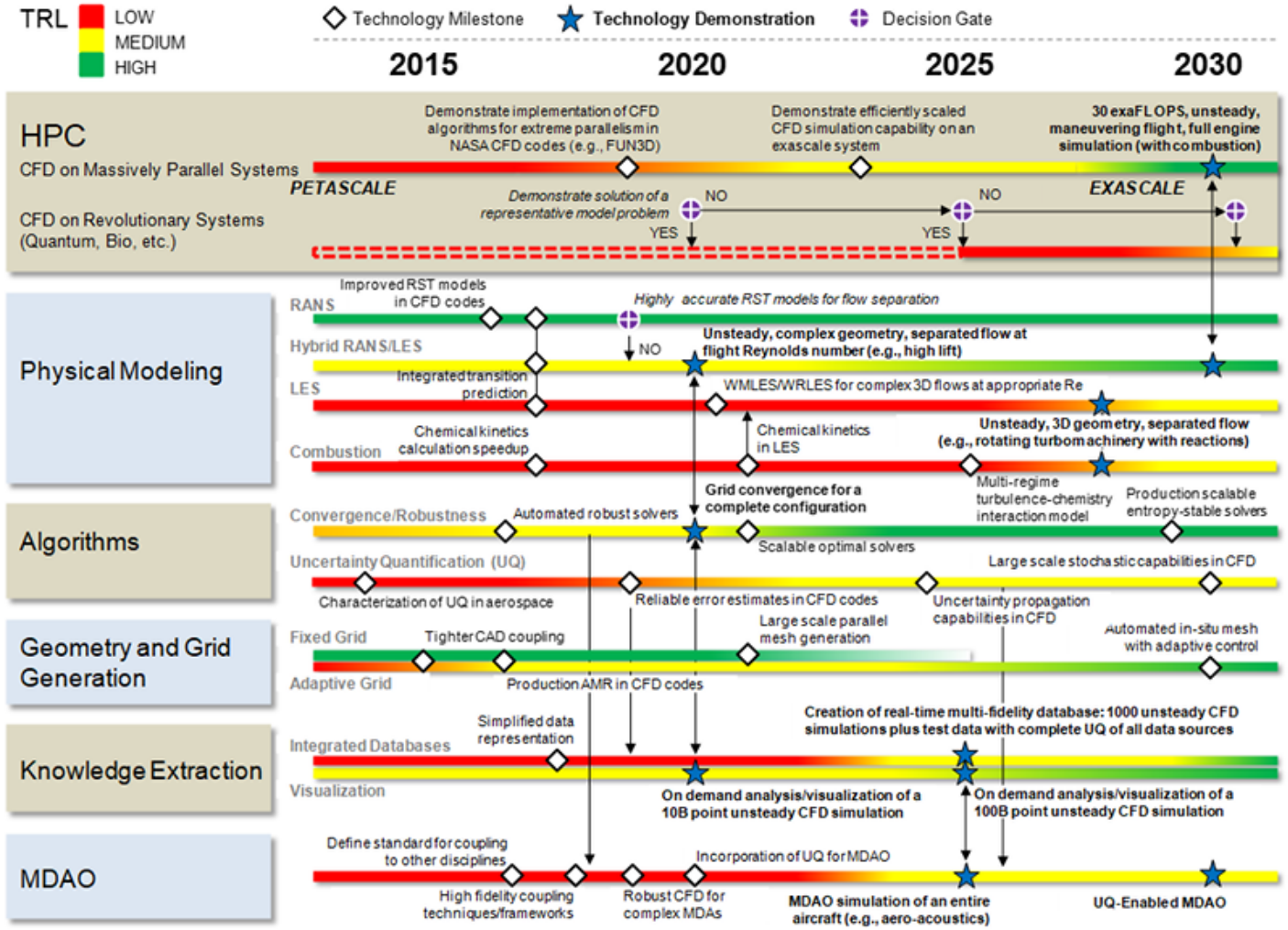
Mohammad Zubair
Old Dominion University

CFD Vision 2030 Study



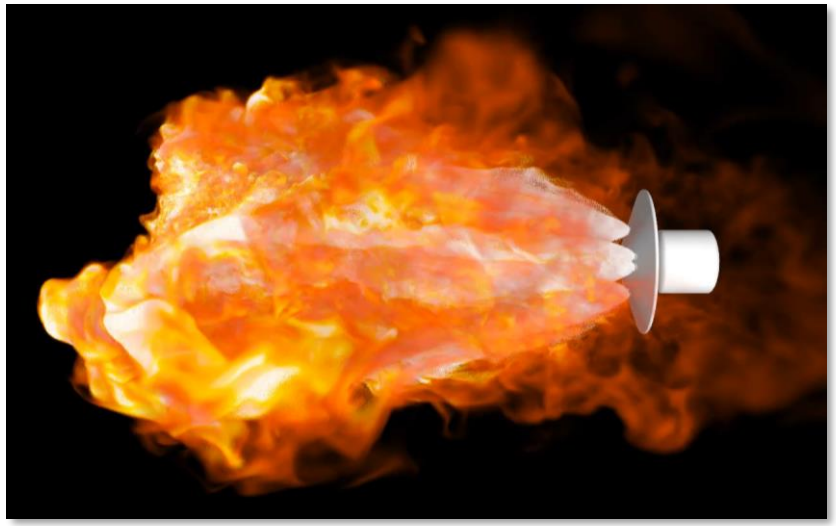
NASA/CR-2014-218178

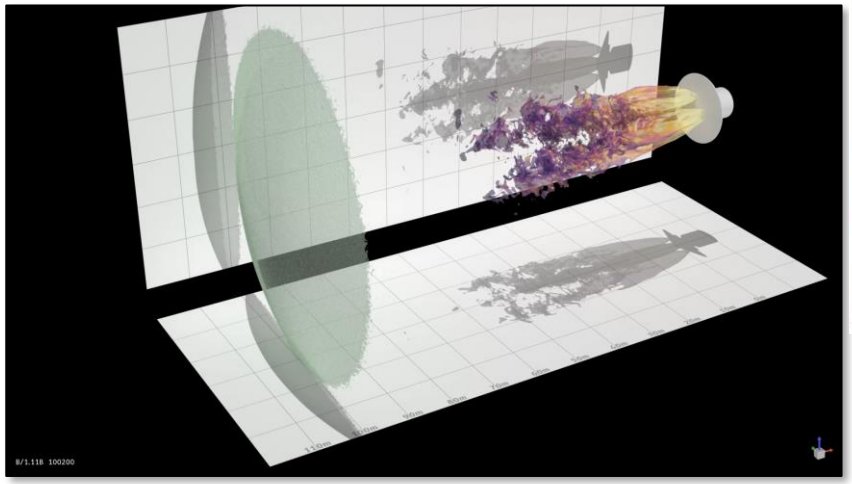
See http://www.cfd2030.com

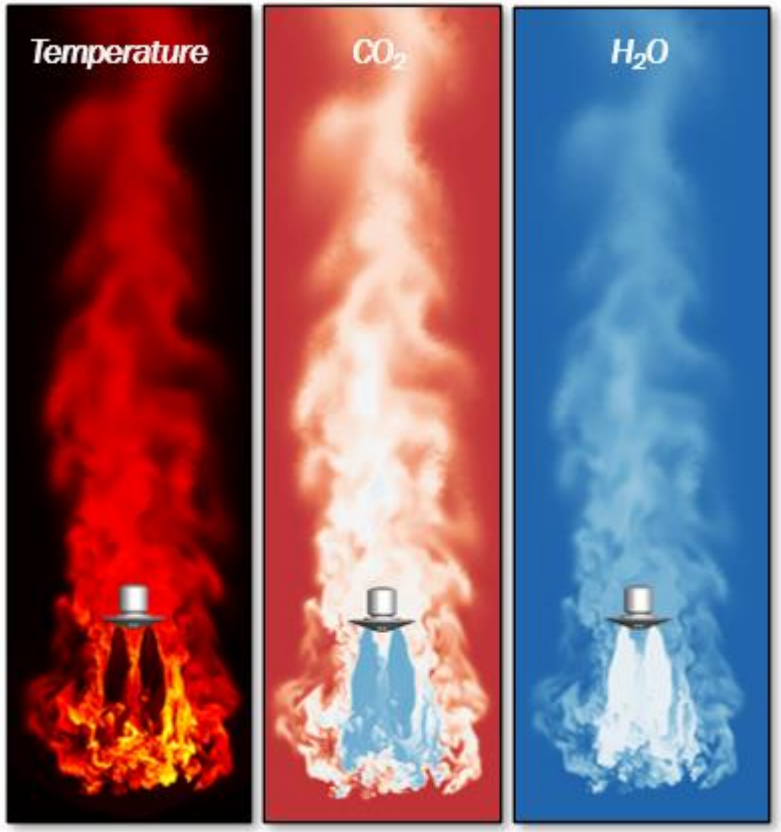


Recent Summit Campaigns

- Summit Early Science, INCITE campaigns for simulations of 16-meter diameter human-scale Mars lander with O_2/CH_4 combustion in Martian CO_2 atmosphere
- DES of 10 species/19 reactions, 7B elements, seconds of real time
- Runs on 15,912 V100s with approximate throughput of several million CPU cores
- Big data: 90 GB of asynchronous I/O every 30 seconds for 2 days yields ~1 PB per run; 60 TB/day migrated from ORNL to NASA Ames
- Stepping stone to CFD 2030 exascale milestones







This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-000R22725.

Governing Equations and FUN3D Background

- Conservation of species, momentum, energies, and turbulence variables
- Two-temperature model available for thermal nonequilibrium
- Variable species, energies, and turbulence equations
- Node-based finite-volume approach on general unstructured grids
- Fully implicit formulation is used to integrate the equations in time
  - Block sparse linear system **Ax=b**
  - Matrix **A** composed of diagonal and off-diagonal $N_{eq}\,x\,N_{eq}$ blocks
  - Memory and solution time increases as $O\left(N_{eq}^2\right)$
- System solved with multicolor point-implicit approach
- Linear algebra and most other kernels are bound by memory bandwidth

$$\frac{\partial}{\partial t}(\rho y_s) + \frac{\partial}{\partial x_j}(\rho y_s u_j) - \frac{\partial}{\partial x_j}(J_{sj}) = \dot{\omega}_s$$

$$\frac{\partial}{\partial t}(\rho u_i) + \frac{\partial}{\partial x_j}(\rho u_i u_j + p\delta_{ij}) - \frac{\partial}{\partial x_j}(\tau_{ij}) = 0$$

$$\frac{\partial}{\partial t}(\rho E) + \frac{\partial}{\partial x_j}\left((\rho E + p)u_j\right) - \frac{\partial}{\partial x_j}\left(u_k\tau_{kj} + \dot{q}_j + \sum_{s=1}^{N_s} h_s J_{sj}\right) = 0$$

$$\frac{\partial}{\partial t}(\rho E_v) + \frac{\partial}{\partial x_j}(\rho E_v u_j) - \frac{\partial}{\partial x_j}\left(\dot{q}_{V\,j} + \sum_{s=1}^{N_s} h_{V\,s} J_{sj}\right) = S_v$$

$$\frac{\partial}{\partial t}(\rho\tilde{\nu}) + \frac{\partial}{\partial x_j}(\rho\tilde{\nu}u_j) - \frac{\partial}{\partial x_j}\left(\frac{1}{\sigma}\left(\mu\frac{\partial\tilde{\nu}}{\partial x_j} + \sqrt{\rho}\tilde{\nu}\frac{\partial\sqrt{\rho}\tilde{\nu}}{\partial x_j}\right)\right) = S_{\tilde{\nu}}$$

$$\boldsymbol{q} = [\rho\vec{y}_s, \rho\vec{u}, \rho E, \rho E_v, \rho\tilde{\nu}]^T$$

$$\int_V \frac{\partial \boldsymbol{q}}{\partial t} dV + \oint_S (\boldsymbol{F}\cdot\boldsymbol{n}) dS - \int_V \boldsymbol{S} dV = \boldsymbol{0}$$

$$\left[\frac{V}{\Delta\tau}\boldsymbol{I} + \frac{V}{\Delta t}\boldsymbol{I} + \frac{\partial\widehat{\boldsymbol{R}}}{\partial\boldsymbol{q}}\right]\Delta\mathbf{q} = -\mathbf{R}(\mathbf{q}^{n+1,m}) - \frac{V}{\Delta t}(\mathbf{q}^{n+1,m} - \mathbf{q}^{n})$$

$$\mathbf{q}^{n+1,m} = \mathbf{q}^{n+1,m} + \Delta\boldsymbol{q}$$

GPU Implementation: FLUDA

- Pre-processing routines remain on the host
- All PDE kernels (~150) performed on the device
- Minimal data transfer between host/device (mainly scalars)
  - Large data motion only at user-specified frequencies (e.g., restarts, visualization support)
- CUDA-based C++ port of compute kernels in FUN3D
- Currently written using a thin layer above AMD Heterogeneous Interface for Portability (HIP) enabling:
  - NVIDIA GPUs (through native CUDA)
  - AMD GPUs (through HIP)
- Other frameworks/approaches are being considered in the frame of performance portability
  - Aim to reach a high percentage of peak memory bandwidth on key kernels such as linear algebra
- Data structures are identical between CUDA and Fortran contexts
  - Column-major order array layouts
  - GPU “mirror” data structures that match CPU data structures
  - Variable precision is identical to CPU approach
    - FP64 for most variables, with mixed FP16/FP32/FP64 for linear algebra

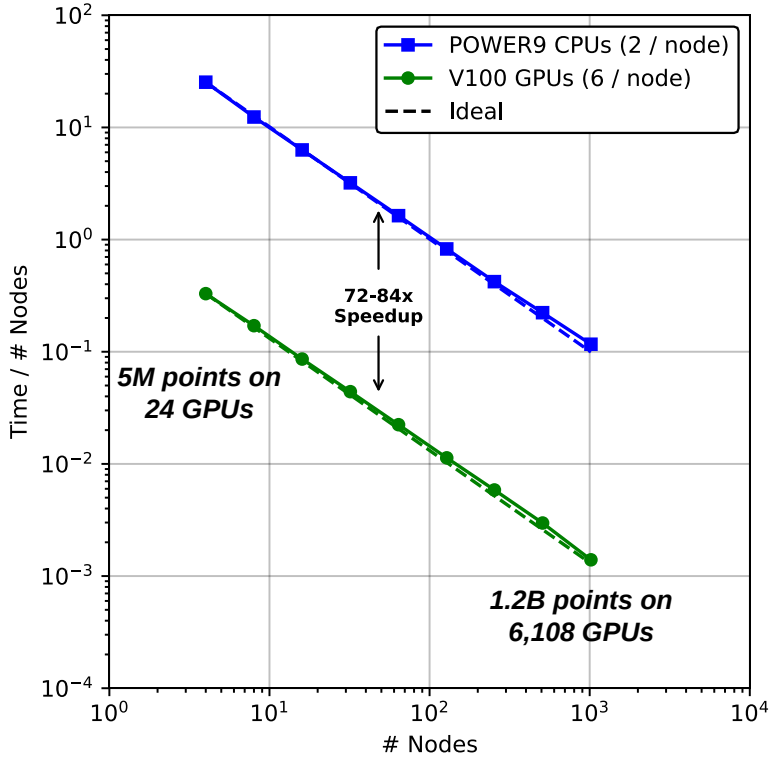
General GPU Optimizations and Performance

- Focus on reduction of kernel state; strategic use of registers and shared memory
- Many kernels employ hierarchical parallelism, especially for chemistry-related kernels
  - Many threads work on an item such as a flux or Jacobian
  - Threads/item and items/block are parameterized
  - Parameters are tuned by an automated process for each device and equation set, which may result in hundreds of different parameter settings for each kernel
  - Tuning has improved performance by over 5x for some kernels

Device-Level Performance (13s/1e/2t)

2 x 64-core AMD 7742	1.0x
NVIDIA V100 32 GB	4.0x
NVIDIA A100 40 GB	7.0x

Scaling on Summit



Race Conditions

- Continuous discretization leads to frequent race conditions when performing edge- and element-based traversals to assemble physics-related terms to the vertices
  - Mixed-precision data requiring FP32 and FP64 updates
- Atomic updates ensure that parallel updates of the same memory location are serialized
  - Collisions: High contention increases cost
  - Bandwidth: L2 cache must be updated: global bandwidth stressor
  - Alternatives (not presented here): Coloring approaches, vertex-based loops
- Atomic update support in hardware: NVIDIA V100+ supports FP32 and FP64; AMD MI100 supports FP32

Consider an edge-loop where a thread is assigned to each edge:

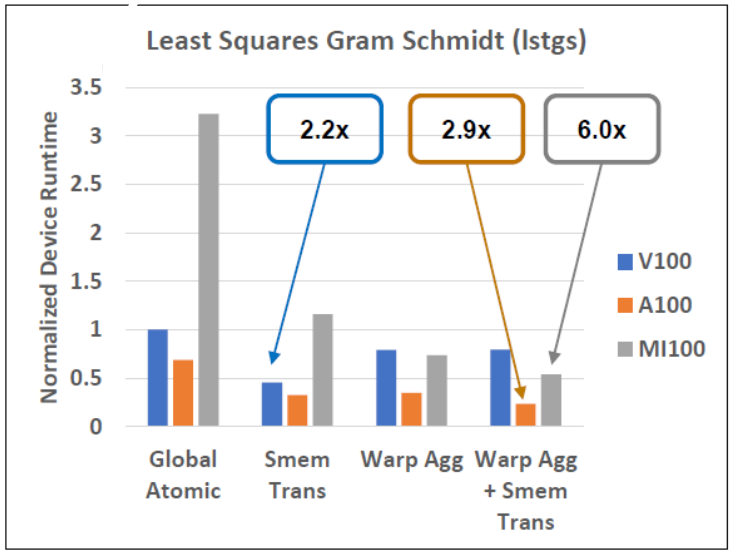
- High probability of threads in same warp updating the same vertex
- Nodes are ordered to improve locality and edge list is sorted by left node index
  - Both increase the atomic collision rate

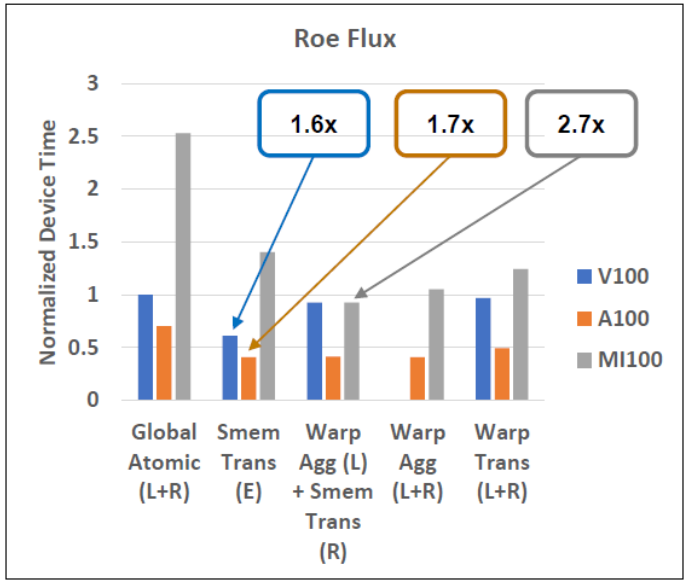
Edge Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**…**
Left node	0	0	0	1	1	1	1	1	2	2	2	…
Right node	3	1	2	3	4	2	5	6	3	9	20	…

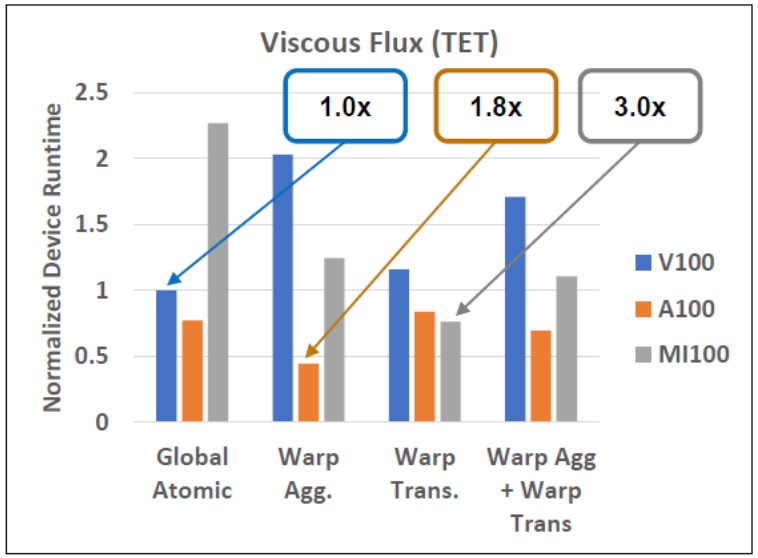
Optimizations Related to Race Conditions

- Improve atomicAdd data access pattern by array transposition
  - Transpose per-thread update values to enable coalesced updates of global vector
  - Two approaches: Shared-memory buffering, warp-level register shuffles (not presented here)
- Reduce atomic update contention with pre-atomic warp aggregation
  - Aggregate common vertex updates at the warp level
  - Register shuffle-based reduction algorithm for matching node indices
  - Issue only a single atomicAdd per unique node index per warp rather than per thread
- All of these approaches (and others) explored in detail in SC21 IA^3 paper:

  Stone, C., et al., "*Accelerating Unstructured-Grid CFD Algorithms on NVIDIA and AMD GPUs*"



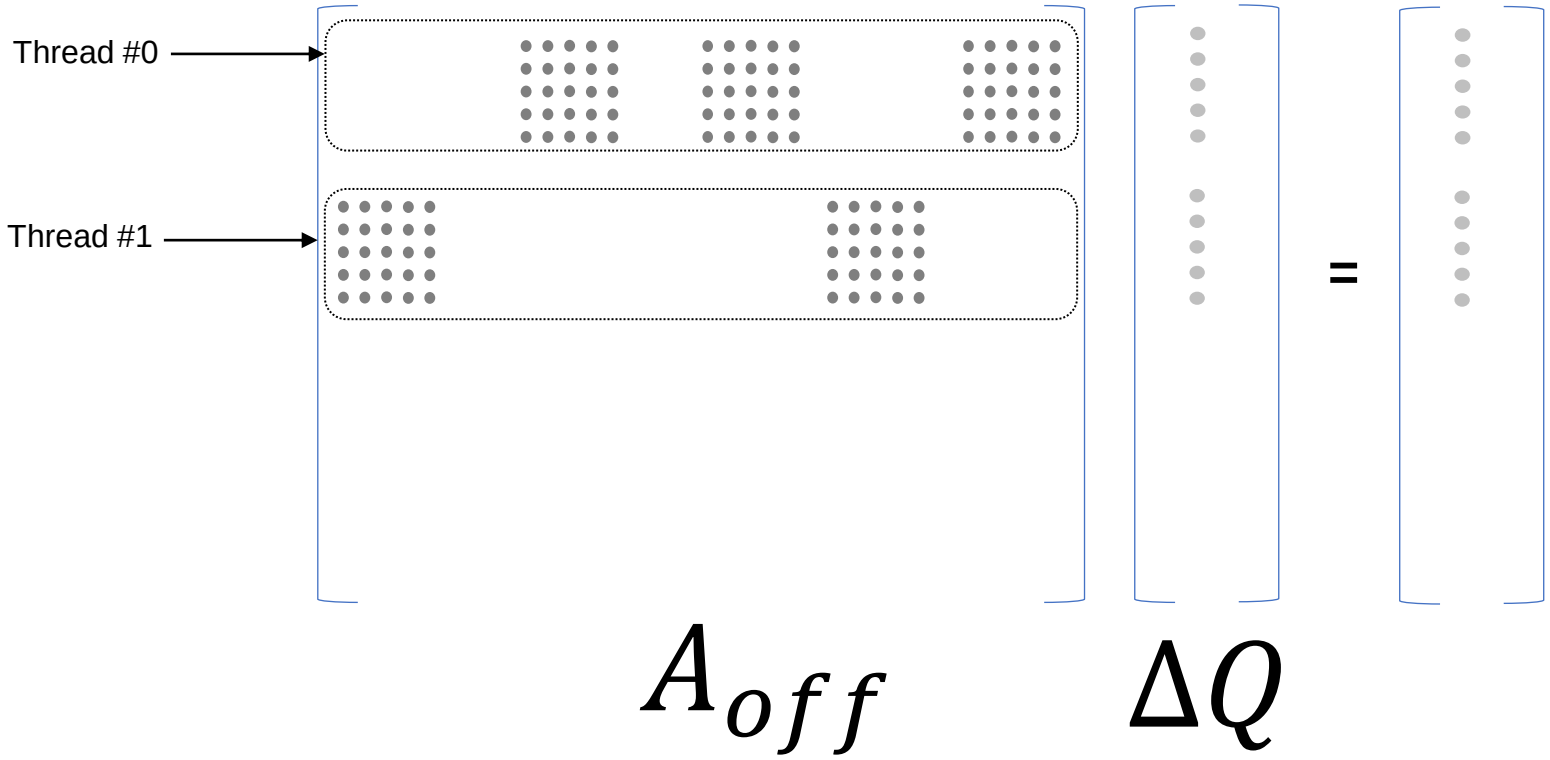




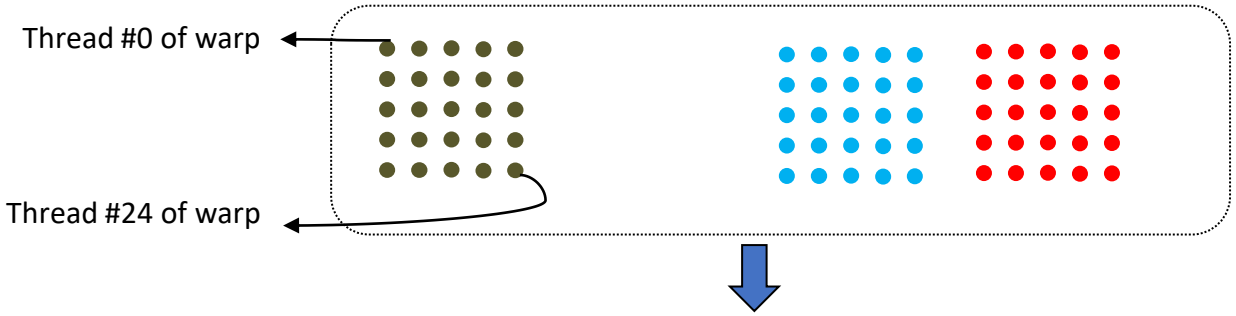
B-SpMV on NVIDIA V100

- Memory access pattern for a naive implementation results in very poor utilization of memory bandwidth
- Consecutive threads are accessing nonconsecutive memory locations
- Wall time is ~500 ms and achieves ~8% of the theoretical peak memory bandwidth of 900 GB/s

Block-Sparse Matrix-Vector Product for 5x5



Optimization of B-SpMV on NVIDIA V100




```
k = threadIdx.x % 5;
L = threadIdx.x / 5;

for (j=istart-1; j < iend; j++) {
    colid = jam[j];
    fk += A_OFF(k,l,j)*DQ(l,colid-1);
}
//save partial aggregation in shared memory
sm_f[k][l][threadIdx.y] = fk;
.
.
```

Output is a 5x5 block of partial terms

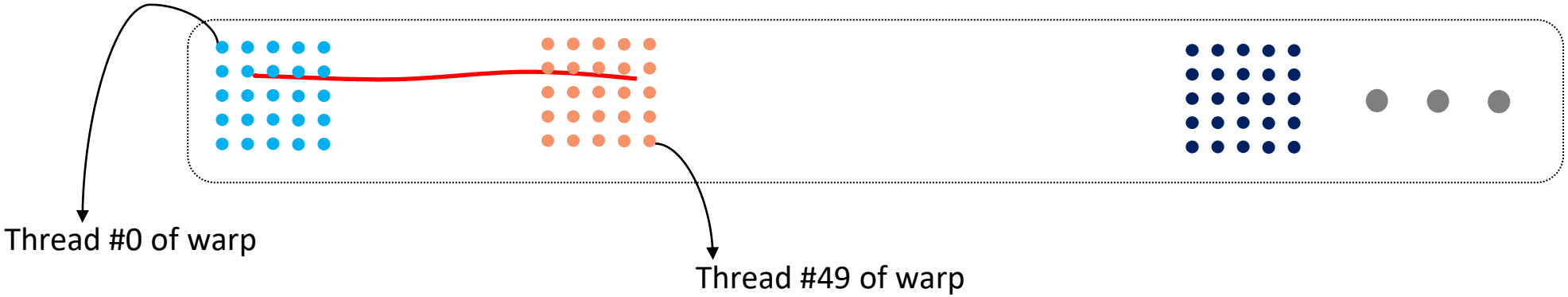
Use one thread to aggregate 5 subcolumns in shared memory into a single column

- Map warp to a 5x5 block
- Coalesced loads
  - Note only 25 threads are active
- Wall time is ~48 ms and achieves 79% of peak memory bandwidth
- 10x performance improvement

Alternative: Avoid shared memory, use shuffle to aggregate

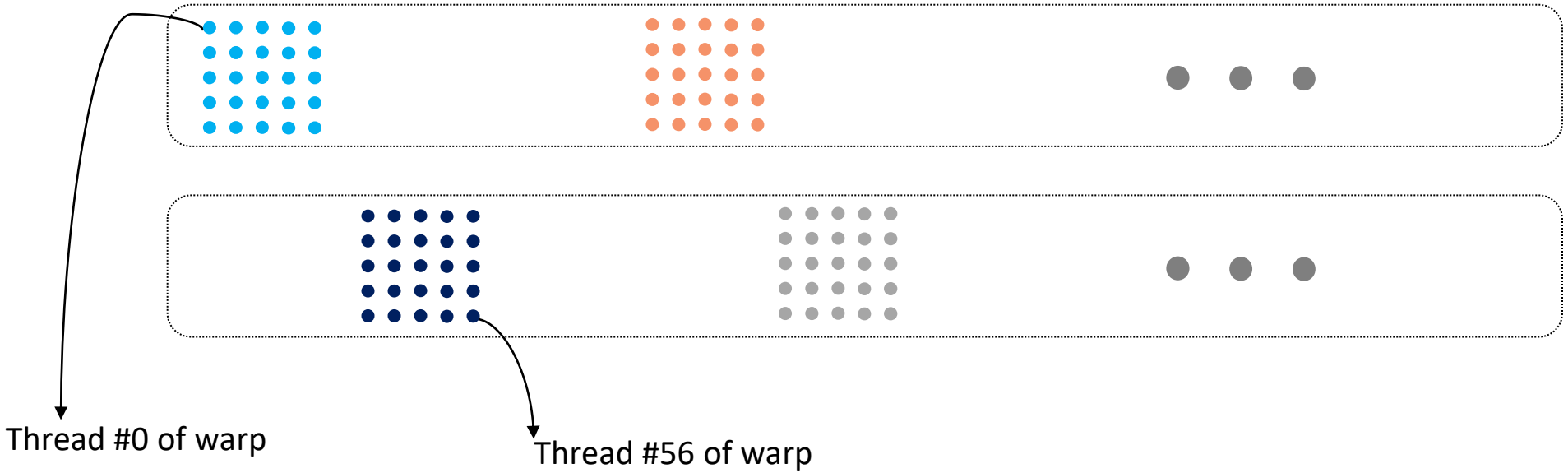
```
// Reduction along the subcolumns
f1 = fk;
f1 = f1 + __shfl_sync(0xffffffff), fk, k + 1 * 5);
f1 = f1 + __shfl_sync(0xffffffff), fk, k + 2 * 5);
f1 = f1 + __shfl_sync(0xffffffff), fk, k + 3 * 5);
f1 = f1 + __shfl_sync(0xffffffff), fk, k + 4 * 5);
```

B-SpMV on AMD GPUs



- On AMD GPUs, warp size is 64
- Consider assigning a warp to two consecutive 5x5 blocks, which achieves coalesced loads

B-SpMV on AMD GPUs



- Performance can be improved by using a warp to process a block in two consecutive rows
- Note consecutive rows may not have identical number of blocks; not all 50 threads will always be active

B-SpMV Performance Summary

Architecture	Programming Model	Wall Time (ms)	Memory Bandwidth (Percentage of Peak in GB/s)
Intel Xeon 6148	AVX512 intrinsics	166.0	78.3% of 256
Marvell ThunderX2	Neon intrinsics	167.0	62.6% of 318
AMD MI50	HIP	64.5	51.3% of 1,024
AMD MI100	HIP	43.8	64.9% of 1,200
NVIDIA V100	CUDA	48.0	78.6% of 900
NVIDIA A100	CUDA	27.2	81.2% of 1,555

- The implementation for the NVIDIA A100 also leverages asynchronous copies directly from HBM and L2-persistence of the vector as described in SC21 MCHPC paper:

  Zubair, M., et al., "*Memory Optimizations for Sparse Linear Algebra on GPU Hardware*"
- SYCL implementation has also been performed

Summary

- Memory performance is critical for the unstructured-grid CFD application used here
- Shared-memory transposition prior to atomicAdd of flux vectors provides consistent performance benefits on all three GPUs shown
  - 1.6-2.2x (V100), 1.7-2.2x (A100), 1.8-2.8x (MI100)
- Warp aggregation improves performance on A100 and MI100 GPUs
  - 1.4-1.8x (A100), 1.4-4.4x (MI100)
  - Consistently poor performance on V100: 0.5-1.3x
- The performance of block-sparse matrix-vector products varies directly with achieved memory bandwidth and is highly dependent on threading strategies and other factors
  - New features on A100 enable additional performance gains (asynchronous copies, L2-persistence)

Thank you for having us!